Research report 125

CLUSTER PROGRAMMING LANGUAGE: DEFINITION AND USER MANUAL

Rolf M Howarth & Nick D Francis

(RR125)

The Cluster Programming Language, or CPL, is a high-level array programming language, used to program the lower levels of the Warwick Pyramid Machine. It is designed to closely match the structure of the architecture, in order to provide a convenient model for users of the machine and to permit an efficient implementation of the language. Because of the nature of the architecture, the language combines both SIMD and MIMD style parallel programming constructs. In this report the language is defined, with several complete examples of code being given, and use of an early interpreter implementation is also described.

Department of Computer Science
University of Warwick
Coventry CV4 7AL
United Kingdom

July 1988

Cluster Programming Language: Definition and User Manual

Rolf M. Howarth and Nick D. Francis
Department of Computer Science
University of Warwick
Coventry CV4 7AL
E-mail: vlsi@flame.warwick.ac.uk

1. Introduction

The Cluster Programming Language, or CPL, is a high-level array programming language, used to program the lower levels of the Warwick Pyramid Machine (see below). It is designed to closely match the structure of the architecture, in order to provide a convenient model for users of the machine and to permit an efficient implementation of the language. Because of the nature of the architecture, the language combines both SIMD and MIMD style parallel programming constructs. In this report the language is defined, with several complete examples of code being given. Use of an early interpreter implementation is also described.

2. Hardware model

The Warwick Pyramid Machine[1,2] is a parallel architecture designed for real-time image and signal processing, with particular emphasis on the transformation from iconic to symbolic data representations. It consists of several heterogeneous layers: a fine grain array of single bit processors (one per pixel) for numeric processing, an intermediate control layer, and a large grain array of transputers for symbolic processing (*fig. 1*). At each level the type and granularity of processor is chosen to match the structure of the data and the types of operation to be performed on it.

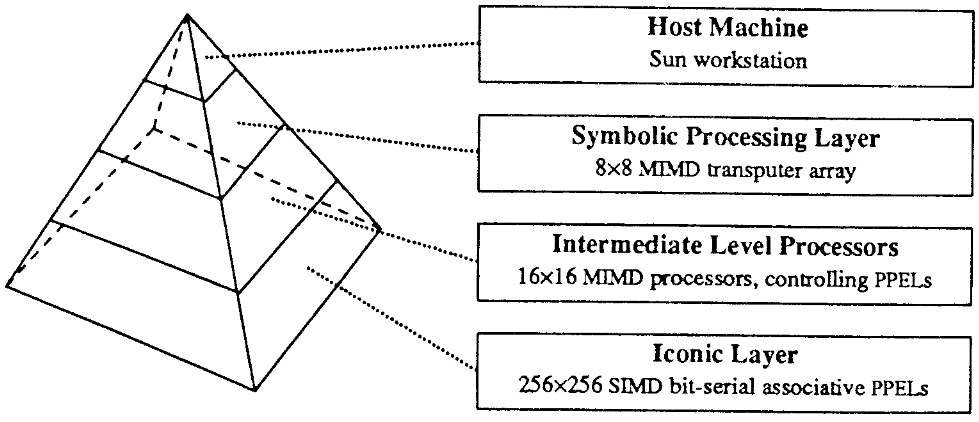

Host Machine
Sun workstation
Symbolic Processing Layer
8x8 MIMD transputer array
Intermediate Level Processors
16x16 MIMD processors, controlling PPELs
Iconic Layer
256x256 SIMD bit-serial associative PPELs

Fig. 1 Warwick Pyramid Machine

For low and intermediate level image processing the array of bit-serial associative Pixel Processing Elements (PPELs) is used, arranged in clusters of 16×16 PPELs each with an associated Intermediate Level Processor (ILP) controlling the cluster. The PPELs within a cluster operate in broadcast instruction SIMD mode, but the ILPs themselves are independent, providing local control. Together these two levels form a multi-SIMD machine.

In this document we restrict our attention to the ILP/PPEL cluster level of the machine, as the transputer level above is programmed separately (in either occam or parallel C) and runs more or less autonomously.

A cluster should always be regarded as a whole - as a single processor capable of performing either scalar (ILP) or vector (PPEL) operations:

2.1 Scalar processor

The ILP is a general purpose 16-bit processor, implemented as a micro-programmable bit-slice processor, with instructions to access the PPEL array wired into its instruction set (one bit in the microcode word selects whether the instruction is an ILP or a PPEL instruction). The PPEL array is in effect a co-processor to the main ILP.

Each ILP/PPEL cluster can be regarded as a conventional SIMD machine, but the machine as a whole contains many such clusters, and is capable of MSIMD operation since each cluster can operate independently. When it is necessary to operate several clusters as a single entity, adjacent ILPs must be synchronised in order that PPEL communication across the cluster boundaries performs as expected.

The dual-ported ILP data memory is shared between the ILP and the transputer above it in the pyramid, and is used for communication between the two. The ILP microcode store is also a dual-port RAM. The transputer writes to it at boot time to load either a complete application or a standard set of library routines callable from an application running on the transputers (this library might include a multi-bit multiply command involving many hundreds of one bit PPEL instructions, or a routine to perform a Sobel edge detection convolution, for example). These routines are usually written in CPL, which is then compiled into ILP microcode.

2.2 Vector processor

The PPELs are simple but numerous processors, each with a bit-serial ALU, 7 flag bits used for storing intermediate results, and (currently) 256 bits of general purpose memory.

The PPELs operate in SIMD mode (each processor executing the same instruction but on different data), with instructions broadcast over the array on a common bus. PPEL operations may be executed unconditionally or be restricted to those PPELs in a cluster which have the active flag 'A' set. In addition the ILP can address an individual PPEL or group of PPELs by placing a mask on the address mesh, described in the next section.

The 256 bits of memory for each processor can be divided into arbitrary sized words (for example, it may be used as a collection of 8-bit grey-scale images, floating point values, single bit flags, and intermediate arithmetical results of varying lengths).

The ALU performs arbitrary logic functions mapping three input bits to two output bits. In each instruction cycle two source operand bits *i* and *j* (from the flags, from main memory, or from a neighbouring PPEL) together with the accumulator flag F are read into the ALU, combined by some logical function, and two result bits written out: one back to the F flag and one to another destination *d*. This operation is performed by all enabled PPELs within a cluster concurrently. By making use of the flag bit it is possible to perform simple multi-bit operations such as addition at the rate of one bit per instruction cycle.

function code	cond *a k v*	src *i*	src *j*	dest	o/p sel	address
16	3	4	4	3	3	8

Table 1 PPEL instruction op-code

Each of the argument fields in the op-code specifies either one of the flag bits, a value from the register file, or the output from an adjacent PPEL. It is possible for both source and destination to lie in the register file, but then they must be the same bit as there is only a single address field. The output select sub-field specifies what data is output to neighbouring PPELs (so, for example, all the PPELs could copy the F flag from the PPEL to their north into their Y flag, at the same time passing their F flag on to the south).

X	Y	Z	T	U	V	A

F

M0	M1	M2	M3 ...	M255

X	Output bit; passes results to the open collector H and V response circuitry.
F	ALU flag; carries over results during multi-bit operations.
Y, Z, T, U, V	Spare bits; used for intermediate results, saving 'A' etc.
A	Active bit; if the condition bit is set in the opcode then only those processors whose active bit is set perform the operation, otherwise 'A' may be used as another general purpose bit.
M	General purpose 256-bit register file.

Table 2 PPEL flags and memory

2.3 PPEL/ILP communication

The PPELs are connected together in the form of a mesh with 8-way nearest neighbour connectivity. Data can be read in from these neighbour lines during normal PPEL instruction cycles as mentioned earlier. In addition to this there are several mechanisms for communication between the PPELs and the ILP, utilised by special instructions at the ILP level.

Associative feedback to the controlling ILP is provided via the PPEL X output flag. An open collector wired-OR circuit gives a some/none response over the whole cluster. At the same time a fast adder tree provides the ILP with a count of the number of responding PPELs in a cluster.

To facilitate reading data into and out of the array the PPELs are also connected on a grid of addressing lines with a wired-OR response capability. These lines, running along all the rows and columns in a cluster with an encoder at the two edges, enable individual pixels to be located quickly (*fig. 2*). The ILPs also have direct access to the lines, giving them the ability to establish whether more than one PPEL is responding.

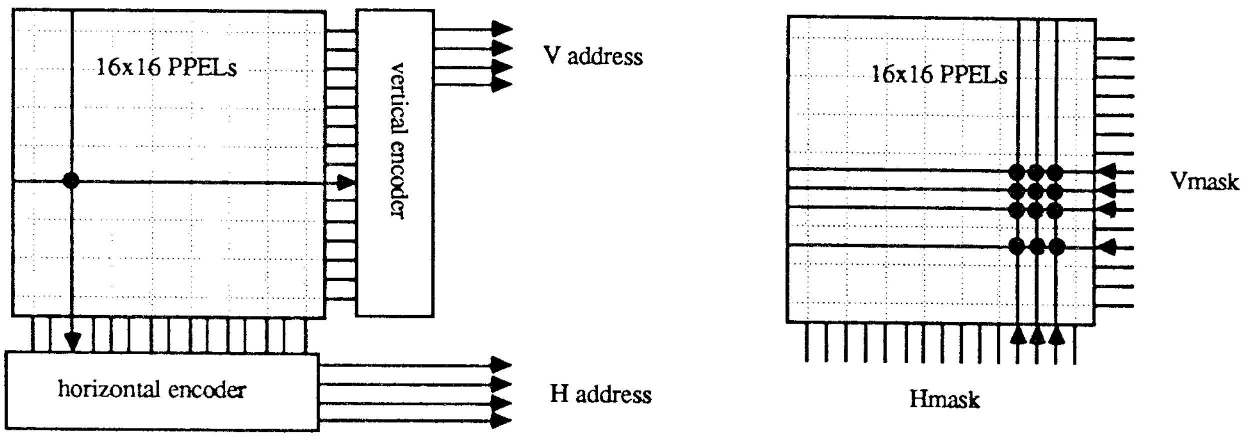


Fig. 2 Use of the H/V responder mesh

The lines are bidirectional, with the horizontal and vertical directions being controlled by the *h* and *v* condition bits in the PPEL instruction word. This allows the ILP to directly address an individual PPEL or group of PPELs, using the cluster like a RAM. Another application is to write on one and read from the other, enabling a single column say, and then reading the 16 bits from that column in parallel.

2.4 ILP/ILP and ILP/transputer issues

The scenario in which ILP programs are run is that of procedures being called from the controlling transputer. This might be achieved by means of a small operating system kernel running on each ILP and implementing a procedure call protocol via the dual-ported RAM shared with the transputer. When an ILP procedure finishes, control returns to this kernel, which makes any result available to the transputer and awaits the next procedure call. An alternative approach is to have the transputer directly set up a stack frame in the ILP memory, as if a call were being carried out by the ILP, and then load the ILP instruction counter with the start address.

For reasons of simplicity, the same program (ie. set of callable procedures) is loaded into all the ILPs. There is no requirement, however, that different transputers should call the same procedure together, which permits the ILPs to run independently.

When necessary, synchronisation between ILPs is performed via a number of open collector sync channels. The basic mechanism is that those ILPs wishing to synchronise first all pull the sync channel low, ie. inactive. They can then perform individual operations. When an ILP finishes, it stops forcing the channel low and idles, waiting for it to become active again. Eventually more and more ILPs will finish and become ready, until the last one goes ready, when the sync line floats high and all the ILPs may continue together, in sync.

There are several sync channels available, permitting independent groups of ILPs to synchronise themselves. One transputer in the system acts as master, allocating this resource (sync channels) on request to those transputers wishing to synchronise their ILPs. Typically one of the parameters in a transputer to ILP call will be the number of a sync channel to use. To protect against the possibility that one of the ILPs in a group wishing to synchronise becomes ready before the other ILPs have initialised the sync operation by pulling the line low, the controlling transputer itself forces the sync channel inactive until it receives confirmation that all the ILPs using that channel have initialised.

Note that we have only described the low level mechanism of ILP synchronisation. Responsibility for breaking down an application into tasks for the ILPs to perform concurrently and deciding when synchronisation needs to take place, and between which ILPs, lies with the transputer array. The master transputer plays a key role in this, making use, amongst other things, of a global some/none and response count over the entire ILP array (an extension of the circuitry within each cluster), but a discussion of how the transputer level is programmed lies outside the scope of this document.

An alternative means of achieving purely local synchronisation is by using the ILP to ILP communication primitives 'send' and 'receive'. These use a rendezvous mechanism, whereby two adjacent ILPs must be ready at the same time, one to send and one to receive, before a communication is able to take place. When necessary, data can also be transferred to or from the controlling transputer in the same way.

3. CPL language definition

When using CPL the programmer's model closely matches the hardware. Scalar operations, including ILP integer arithmetic, function calls and loop control, use a simple conventional syntax similar to Pascal or C. In addition to this standard core, though, there are commands to perform a vector operation in parallel over all the PPELs in the cluster, as well as operands to read data in and out of the PPEL array which can be used in ILP expressions.

Because there are these two different kinds of processor within a cluster, it is helpful always to make a clear distinction between ILP and PPEL instructions in a CPL program. For example, the '*if*' instruction is the familiar conditional at the ILP level, while '*where*' is used to restrict a PPEL operation to certain pixels within a cluster. These operations, as well as calculations and assignments and so on, are analogous at the two levels, but if the syntax of the language were to hide the distinction between similar operations on different processors it would undoubtedly lead to confusion.

We therefore introduce the basic ILP and PPEL operations in separate sections, followed by a description of interfacing between the two. In the following definitions we use bold for reserved words or symbols, and italic for other syntax elements. Example code is in a sanserif font.

A program consists of statements, usually one per line, though a semicolon can be used to separate multiple statements on the same line. Spaces between tokens are needed where ambiguities might otherwise arise. Extra spaces and blank lines are ignored. Everything between a '#' and the end of a line is ignored as a comment.

A. Conventional scalar programming

This section describes those ILP operations which are familiar from normal sequential, procedural languages.

3.1 ILP variables and expressions

ILP variables are 16 bit signed integers. They are referred to by symbols, consisting of upper and lower case alphabetics and the underscore character, which must be declared before use:

int: *ilpvar, ...*

An ILP expression may consist of just a single term, either an ILP variable, an integer constant, a function return value, or some other special term (such as 'any', see 3.8). These basic terms can also be combined using the arithmetic and logical operators given in *table 3* (based on those of the C language[7]). Parentheses may be used to alter the precedence of operators in an expression. Constants are usually in decimal, but may be in binary or hexadecimal if preceded by '0b' or '0x' respectively.

Expressions may be evaluated and the result assigned to an ILP variable:

ilpvar **:=** *ilpexpr*

For example,

```
int: average, x
average := (count( )+x)/2
```

Arrays of integers can also be defined. Conventional square bracket notation is used, eg.

```
int: a[10]                          # 10 element array, a[0] to a[9]
x := a[2]                           # refer to the third element
```

~	bitwise not (unary)
!	logical not (unary)
−	negation (unary)
* / %	multiply/divide/modulus
+ −	add/subtract
<< >>	left/right shift
&	bitwise and
^	bitwise xor
|	bitwise or
< <= > >=	comparison
!= =	(in)equality
&& ||	logical and/or

Table 3 ILP operator precedence

3.2 Control constructs

Various control constructs are available. Remember that in general many ILPs are running the same program concurrently, so in a conditional branch or loop some ILPs can be executing one branch while some are following the other. See section 3.9 for notes on synchronising the operation of multiple ILPs.

The syntax of a conditional branch is

```
if expr then
        ... stmts ...                    # carried out if expr is true
[else
        ... stmts ...]                   # carried out if expr is false
endif.
```

If there is only one statement in the body it may be placed on the same line as the 'if', and the 'endif' be omitted, for example

```
if x<0 then x := -x
```

The simplest way to implement a loop is with the 'for..next' statement. The two expressions are evaluated once at the start to give the lower and upper bounds of the loop. On each iteration the loop variable is incremented by one, or by the step size if given (which may be negative, in which case the loop counts down), until it reaches the upper bound.

```
for var = expr to expr [step expr]
        ... stmts ...
next var.
```

There are two forms of 'while' loop, the standard

```
while expr do
        ...                              # carry out body of loop if expr is true
endwhile
```

and with the condition at the end, so the loop is always carried out at least once

```
do
        ...
while expr.
```

A generalised loop in the manner of Ada exists too, where the condition, or conditions, can occur at any point within the loop. The '**exit**' statement causes a jump to the end of the loop, and can also be used to break out of 'while' or 'for' loops.

```
loop
        ...
        exit [when expr]                    # exit when condition is true
        ...
endloop.
```

3.3 Program structure: functions

A CPL program consists of a number of function or procedure definitions. As in C there is no distinction between functions returning a value and procedures which do not. Unlike C however, there is no 'main' procedure where execution always starts, since it is up to the controlling transputer to initiate execution of code on a cluster by calling any procedure it choses. Functions may be called by the transputer or from within another CPL function, but the syntax remains the same.

```
func name ( [parameter list] )
        ... body of function ...
        [return expr]                       # optional (integer) return value
endfunc.
```

The parameter list specifies the type of formal parameters being passed to the function, eg.

func sobel (int: n; bitplane: image, result).

Variables defined in the parameter list or within a function are local to that function. All variables, both ints and bitplanes (see below), that are declared outside of a function are static and global, however, and their value is accessible from within any function.

B. PPEL array programming

3.4 Bitplanes

The basic data types operated on by the PPELs are single bits. Apart from the 8 flag bits X, Y, Z, T, U, V, A and F, each PPEL has a number of bits of main memory, currently 256 bits, addressed M0 ... M255. Contiguous bits of this memory are frequently grouped together for multi-bit operations. For example, M10,11,12,13 may be used as an accumulator, containing one 4-bit number. Data is stored least significant bit first (ie. M10 = lsb, M13 = msb).

Since it is inconvenient to keep track of these locations by explicit address, the language allows named PPEL variables. The statement

bitplane: flag, image[8]

for example, allocates a single bit (in each PPEL across the plane) to a token *flag*, and 8 contiguous bits to a token *image*. In this example *image* might be placed at M10..M17; this would then be accessed using *image* to refer to all 8 bits, *image[1]* to refer to M11, *image[3..5]* to refer to the 3 bits M13..M15 etc.

While it is possible to use explicit memory addresses when special circumstances demand it, and also to access all the flag bits, their use is to be discouraged as no knowledge should be assumed about which locations the compiler may use. The flags X, Y and Z are freely available for use in a CPL program (where X is used to signal a response from the PPEL to the ILP), but the others are used by the compiler for PPEL expression evaluation, as carry and overflow flags, to save the state of A, and so on.

Bitplanes are allocated automatically on a PPEL data stack. When a CPL function returns, the memory used as bitplanes within that function is freed again. Internally a bitplane is really a pointer to some PPEL memory and a count of bits. Normally a 'bitplane' statement allocates the appropriate number of bits in PPEL memory, initialises them to zero, and associates the address and size of the bitplane with the symbol *name*. An alternative use is just to use the address of an existing bitplane, by placing the bitplane at a given address using '='.

*[***signed***] bitplane: name [[size]] [=ppelvar], ...*

For example, if we assume that free PPEL memory (the stack) currently starts at M100, the statement

```
bitplane: image[8]=M16, buffer[8], msb=buffer[7], low[4]=buffer, flag
```

would result in the following bitplanes being defined

```
image   M16..M23
buffer  M100..M107
msb     M107
low     M100..M103
flag    M108.
```

As bitplanes are allocated dynamically their size may be given by an ILP expression calculated at run time.

Bitplanes are normally unsigned, unless a bitplane declaration is preceded with the reserved word 'signed' in which case the most significant bit is used as a sign bit and two's complement arithmetic is used.

3.5 PPEL expressions

Calculations over the array are described by PPEL expressions. When an expression is evaluated it returns a bitplane value which may then be assigned to a PPEL variable or flag:

ppelvar <– ppelexpr.

PPEL expressions consist of bitplanes and numeric constants combined using various arithmetical and logical operators. Since ILP and PPEL expressions operate on different data types a different set of operators is used, but the function and precedence rules for these operators is identical to the corresponding ILP operators (refer to *table 3*).

Arithmetic	*Logical*	*Comparison*	
PLUS	NOT	GT	(>)
MINUS	AND	LT	(<)
TIMES	OR	EQ	(=)
DIVIDE	XOR	NE	(≠)
	SHIFT>	LE	(≤)
	SHIFT<	GE	(≥)

Table 4 PPEL operators

Because PPEL calculations involve variable numbers of bits, the precision used when evaluating an expression is derived from the sizes of the bitplanes used in the expression. The minimum number of bits to guarantee correct evaluation is used. If the number of bits in the variable being assigned to differs from that of the expression being evaluated, the result is truncated or null-padded / sign-extended accordingly (and the F flag is set if the result is too large to fit).

To ensure that PPEL expressions can be evaluated to the appropriate precision the size of a bitplane is always associated with the bitplane itself. During a function call the size of a bitplane is passed along with its address (bitplane parameters are always passed by reference), which allows one to pass bitplanes of arbitrary size to a function, and to write functions which work correctly whatever the precision of their parameters are. The '**sizeof(***bitplane***)**' operator can be used to determine the number of bits in a bitplane.

This default number of bits used when evaluating expressions can be reduced by selecting a slice or subset of a bitplane using the square bracket notation. Start and end bits (with bit 0 = lsb) are specified, where a missing end means take the range up to the end of the bitplane.

A PPEL 'constant' is constant across the cluster. It may be a decimal, binary or hexadecimal constant, or it may be given by the value of an ILP expression if the '$' type conversion operator is used (see 3.8). PPEL constants need not necessarily be fixed at compile time, so long as a single value to be used across the whole cluster is given at the ILP level.

Examples of PPEL expressions

```
num       <- 0                              # Decimal constant
num[0..4] <- 0b10110                        # Binary constant
num[1..]  <- 0xA                            # Hexadecimal constant
num       <- image MINUS $offset
flag      <- NOT ( Z AND (num[0..4] GT 4))
num       <- $(count() + 1)                 # ILP expression
```

3.6 Accessing neighbouring PPELs

To access data from one of the 8 adjacent neighbours the notation *dir:reg* can be used in PPEL expressions, where *dir* is one of N, NE, E, SE, S, SW, W, NW. It may also be of the form '*%ilpexpr*', in which case the ILP expression must evaluate to a number in the range 0 .. 7 which is used as the direction (where N is 0, NE is 1, etc.)

```
image <- N:image                # shift entire image south by one pixel
temp <- %i:num                  # read from neighbour given by variable 'i'
```

One frequently wants to perform an operation over all the neighbours of a PPEL, so the special notation

assoc_op **all:***bitplane*

can be used in expressions with one of the operators SUM (or PLUS), AND, OR and XOR, eg.

```
flag <- OR all:X
```

is equivalent to the much more cumbersome (and less efficient, because in the former case the compiler is able to produce optimised code)

```
flag <- N:X OR NE:X OR E:X OR ...
```

To combine the four direct neighbours only (N, E, S and W) use '**all4**'.

3.7 Conditional operations

The PPELs are SIMD processors and the same instruction is broadcast to all the PPELs within a cluster. Simple conditional operations, restricting a PPEL instruction to those PPELs that have the condition flag 'A' set, can be performed by appending '?' to the instruction, eg.

```
num <- num PLUS 1 ?
```

This will increment the bitplane *num* on just those PPELs whose condition bit is set. Note that this instruction will still take the same number of cycles as an unconditional instruction even if no PPELs are enabled and execute it.

By default conditions act on the A register, but an instruction can also be conditional on the H or V mask (described in 2.3). The '?' modifier may be followed by the letters A, H or V to indicate the condition or conditions, eg.

```
hmask := 0x10
Y <- 1 ?H                       # set all Y's in column 4
vmask := 0x80
X <- 1 ?HVA                     # get PPEL at position (4,7) to respond if A is set
```

To make use of conditionals easier the 'where' instruction restricts execution of a block of code to those PPELs that satisfy the conditional expression (ie. those where *expr* evaluates to a non-zero value)

```
where expr
    ... stmts ...
[elsewhere
    ... stmts...]
endwhere.
```

For example

```
where (SUM all:data EQ 0)
    data <- 0                    # thin isolated points (those with 0 neighbours)
endwhere.
```

'Where' statements can be nested, progressively restricting the set of active PPELs.

There are two variations on the 'where' statement that may be used to restrict operations to a single PPEL (specified by its address) or to an area of PPELs (specified by two mask values) respectively.

whereaddr *address*

address is in the range 0 .. n^2-1, where n is the size of a cluster, typically 16.

wheremask *hmask vmask*

hmask and *vmask* are placed on the horizontal and vertical mesh lines (see 2.3).

A 'where' statement with an 'elsewhere' clause is implemented by first broadcasting the initial block of instructions to the PPEL array, then toggling the appropriate condition bit and broadcasting the 'else' portion. Again, the point should be made that execution time for a 'where' conditional is the same even if no PPELs are enabled to perform one or the other block of instructions.

C. Other aspects of CPL

3.8 Interface between ILP and PPELs

There are a number of special terms and predefined functions that can be used within ILP expressions to access the PPEL cluster.

The horizontal and vertical mesh lines are accessed via two ILP registers, which are referred to from CPL using the reserved words '**hmask**' and '**vmask**'. Any value written to these 'variables' is placed on the mesh when a '?H' or '?V' conditional modifier is used. If they are read in an expression then data is read from the mesh. Similarly, '**any**' is a read only pseudo variable, yielding a Boolean value which is true (ie. 1) if any PPELs in the cluster currently have their X responder bit set, or false (0) otherwise.

The function '**firstX()**' returns the address of the responding PPEL with lowest address, or -1 if there are no responders. This uses the output of the priority encoders in *fig. 2* and works by picking the first row, enabling that row and then selecting the first column with a responder. '**count()**' returns a count of the number of responders in the cluster.

The function '**read**(*address, &ppelvar*)' can be used to read data from a particular PPEL (in effect this works by using 'whereaddr' to select the PPEL, getting it to output its data onto the some/none bus one bit at a time, and accumulating the value in an ILP register).

To pass data from the ILP to the PPELs normal PPEL instructions are used but with the parameters specified indirectly by ILP expressions.

The '\$' type casting operator converts an ILP expression to a PPEL numeric constant, and the '%' operator converts an ILP expression to a bitplane address. No assumptions should be made as to how a pointer to a bitplane is stored at the ILP level (information stored includes the address and size of the bitplane, whether it is a flag or in main PPEL memory, and is signed or unsigned). Instead, '&' can be applied to any bitplane to yield its address and size in this internal format. The '%' operator should only be applied to expressions previously obtained using '&'.

'%' is also used to specify a direction indirectly, as in '*%d:bitplane*'. As a convenient abbreviation, one may omit the '&' when referring to directions, so '*for d = N to W step 2*' can be used to step through the four direct neighbours, for example. The loop variable can be used as a subscript directly (where N=0, NE=1, E=2 and so on) to accumulate a result over neighbours.

For symmetry with 'read( )' the function '**write**(*address*, *&ppelvar*, *ilpexpr*)' is provided, though it is directly equivalent to

```
whereaddr (address)
    ppelvar <- $(ilpexpr)
endwhere.
```

Also useful in this context are several 'compiler constants' which give details of the current implementation: **Isize** (the number of ILPs across in the machine), **Myaddr** (the address of this ILP, from 0 to Isize×Isize − 1), and **Psize** (the size of a cluster, in PPELs across).

3.9 ILP synchronisation and communication

The basic mechanism for ILP synchronisation and communication is as described in 2.4.

In normal use the transputers determine which ILPs should synchronise themselves and pass down the numbers of one or more sync channels for the ILPs to use. To perform synchronisation in a CPL program this sync channel must first be initialised, using

```
syncon n                          # channel n allocated by transputer.
```

The actual synchronisation of ILPs using that sync channel then occurs once they have all executed the instruction

```
sync                              # wait until all ILPs are ready.
```

ILP to ILP communication is via occam-like channels, and is indicated by the reserved word 'chan'. Thus

```
chan c ! ilpexpr                  # send word
chan c ? ilpvar                   # receive word
```

where *c* is one of the five (predefined) channels N,S,E,W and UP. For example,

```
chan E ? x
chan UP ! (result+1)
dir := chan E
chan dir ! 1                      # indirect channel specification
```

(This use of '?' should not be confused with PPEL conditional operations.)

4. References

1. R.M. Howarth, "A heterogeneous pyramid array architecture for image understanding", *Research Report 115*, Dept. of Computer Science, University of Warwick, December 1987.

2. G.R. Nudd, R.M. Howarth, T.J. Atherton, N.D. Francis, G.J. Vaudin, and D.W. Walton, "A heterogeneous architecture for parallel image processing," in *Proc. 1988 UK Information Technology Conference*, pp.495-499, Swansea, July 1988.

3. C.C. Weems and S.P. Levitan, "The Image Understanding Architecture," in *Proc. DARPA Image Understanding Workshop*, pp.483-496, February 1987.

4. S. Pass, "The GRID parallel computer system," in *Image Processing System Architectures*, ed. J. Kittler & M.J. Duff, pp. 23-35, Research Studies Press, 1985.

5. M.J.B. Duff and T.J. Fountain (editors), *Cellular Logic Image Processing*, Academic Press, New York, 1986.

6. D.E. Reynolds and G.P. Otto, "CLIP 'Image Processing C' User Manual", *Report No. 82/4*, Image Processing Group, University College London, 1981.

7. B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.

Example 1: Sobel edge detection

```
# Perform the following convolution for horizontal edges
#     -1  -2  -1
#      0   0   0
#      1   2   1
# and the same thing vertically, then combine the two as the sum of mods.
# Illustrates the use of 'sizeof' to work with arbitrary precision data.

func sobel (bitplane:image,result)
        # Store intermediate results with sufficient precision
        bitplane: partial[sizeof(image)+2], temp[sizeof(image)+2]

        temp[1..] <-- image                    # temp <-- image*2+E+W
        temp <-- temp PLUS E:image PLUS W:image

        temp <-- S:temp MINUS N:temp

        where F                                # negate if negative to get absolute value
           temp <-- 0 MINUS temp
        endwhere

        partial <-- temp                       # partial is used to save |Gx|

        temp[1..] <-- image                    # Same thing vertically
        temp[0] <-- 0                          # cleared initially but now contains junk
        temp <-- temp PLUS N:image PLUS S:image

        temp <-- E:temp MINUS W:temp
        where F
           temp <-- 0 MINUS temp
        endwhere

        # Return |Gx| + |Gy|, scaled to fit size of result
        result <-- (partial PLUS temp) SHIFT> (3+sizeof(image)-sizeof(result))
endfunc sobel
```

Example 2: Edge following

```
# This function follows edges, returning a list of the addresses of edge points
# within a cluster. It ignores any intersections.

func simple_edge_follow (bitplane: Image)
    bitplane: Mark, Edge, Result[8]
    int: addr, th, dir
    sobel(Image, Result)                # calculate 8-bit Sobel
    th := threshold(Result)             # calculate threshold
    Edge <-- Result GT $th              # Set Edge if Result > th
    Mark <-- 0
    loop
      X <-- Edge AND (SUM all:Edge EQ 1) # set X if #neighbours = 1, ie. it's an endpoint
      exit when !any                    # exit loop if no endpoints in this cluster
      addr := firstX( )
      whereaddr addr
        Mark <-- 1                      # pick an endpoint to start from
      endwhere
      do
        addr := firstX( )
        chan UP ! addr                  # output one point to the Symbolic layer
        where Mark
          clear Edge                    # reset the point we have read
        endwhere
        X <-- Mark                      # output mark to its neighbours
        F <-- OR all:X                  # test if any of the neighbour inputs are set
        X <-- Edge AND F
        Mark <-- X
      while any
    endloop                             # continue until the data has been read out
endfunc
```

Example 3: Mean and maximum

```
# Calculate local means and maxima within a cluster simultaneously, one bit at
# a time, by iterating starting with the most significant bit.

int: mean, max                              # global variables to return results

func meanmax (bitplane: image)
    bitplane: flag
    int: total, i

    flag <-- 1                              # this PPEL not less than max (so far)
    total := 0
    max := 0

    for i = sizeof(image)-1 to 0 step -1
      X <-- image[i]
      total := 2*total + count( )
        # If flag=0 I've already been discounted as max, so don't output X
      X <-- X AND flag
      max := 2*max + any
        # If some pixel in the cluster has this bit set (any) but my
        #  bit isn't set (X=0) then I can't be a maximal pixel
      if any then flag <-- X                # if 'any & X=0' then reset flag
    next i

    mean := total/(Psize*Psize)
endfunc


# This code fragment illustrates the 'all' notation:

    bitplane: Nnbrs[3]
    Nnbrs <-- 0                             # Use a 3-bit accumulator to count the neighbours
    for dir = N to NW                       # iterate over the 8 neighbours
      Nnbrs <-- Nnbrs PLUS %dir:Edge
    next dir
# is equivalent to

    Nnbrs <-- SUM all:Edge                  # (SUM is a synonym for PLUS)
```

Example 4: Guarded edge thinning

```
# Elliman and Mahmood's thinning algorithm (adapted by N.Francis)
# incorporates extensive guarding to prevent over-erosion

# From D.G.Elliman and A.Mahmood "Towards faster and more shapely thinning",
#   in Parallel Processing for Computer Vision and Display, Leeds, Jan 1988

func thin (bitplane: edge_bit)
    bitplane: count[4], init_count[4], number_set[4], init_flag
    bitplane: neighbour[8], clear_flag, temp, guard[8], any_guard
    int: d

    # Get all neighbours bits and count them
    for d = N to NW
        neighbour[d] <-- %d:edge_bit
    next d
    number_set <-- SUM all:edge_bit

    # Count number of consecutive bits set in neighbour
    init_flag <-- 1                        # Initialise flags
    clear_flag <-- 0
    for d = N to NW
        where neighbour[d]
            count <-- count PLUS 1         # if bit set inc count
            where init_flag
                init_count <-- init_count PLUS 1  # if bit set & not had a zero yet
            endwhere
        elsewhere
            count <-- 0                    # if bit not set clear count
            init_flag <-- 0                #   and init_flag
        endwhere
        clear_flag <-- clear_flag OR (count EQ number_set)
    next d

    # At this point init_count holds the number of consecutive bits set at start
    # count holds the #consecutive at the end of the word
    # clear_flag is set if we found 'number_set' consecutive bits at any stage

    # Now check for wrap around
    where ((init_count PLUS count) EQ number_set)
        clear_flag <-- 1
    endwhere

    # 'clear_flag' is set if all the neighbouring bits are consecutive.

    # We don't want to thin solid regions or endpoints though...
    where (number_set EQ 8) OR (number_set LT 3)
        clear_flag <-- 0
    endwhere

    # Generate guard signals according to table 2 of Elliman and Mahmood

    temp <-- (number_set EQ 3) AND clear_flag
    where (temp AND neighbour[NW])          # 3 neighbours
        guard[NW] <-- 1
    endwhere
    where (temp AND neighbour[SW])
        guard[SW] <-- 1
    endwhere
                                            # 4 or 5 neighbours
    temp <-- (number_set GT 3) AND (number_set LT 6) AND clear_flag
    where (temp AND neighbour[NW] AND neighbour[SW])
        guard[S] <-- 1
    endwhere
    where (temp AND neighbour[SW] AND neighbour[SE])
        guard[W] <-- 1
    endwhere

    temp <-- number_set EQ 6                # 6 neighbours
    where (temp AND NOT neighbour[N])
        guard[S] <-- 1
    endwhere
    where (temp AND NOT neighbour[NE])
        guard[SW] <-- 1
    endwhere
    where (temp AND NOT neighbour[E])
        guard[W] <-- 1
    endwhere

    temp <-- number_set EQ 7                # 7 neighbours
    where (temp AND NOT neighbour[N])
        guard[S] <-- 1
    endwhere
    where (temp AND NOT neighbour[E])
        guard[W] <-- 1
    endwhere
    where (temp AND NOT neighbour[NE])
        guard[S..W] <-- 7                   # set guards S, SW and W
    endwhere
    where (temp AND NOT neighbour[SE])
        guard[W..NW] <-- 3                  # set guards W, NW and N
        guard[N] <-- 1
    endwhere

    # Check for any neighbour guards
    any_guard <-- 0
    for d = N to NW
        any_guard <-- any_guard OR %d:guard[(d+4)%8]   # d+4 calculates opposite direction
    next d

    # If clear_flag and no guards preventing me then clear my pixel
    where (clear_flag AND NOT any_guard)
        edge_bit <-- 0
    endwhere

endfunc thin
```

An earlier implementation: User manual for the 'sim' interpreter

Introduction

Initially the cluster level of the WPM architecture was simulated by means of an interpreter for an early version of the CPL language, running as a single user process under Unix. This version of the language, now referred to as CPL-1, differs from the current definition in several ways.

There are various syntactic changes, chiefly at the PPEL level, reflecting the fact that in CPL-1 these operations were at a lower level, corresponding much more closely to the hardware (see the section on 'Low level PPEL operations' below).

The PPEL memory model that CPL-1 assumed differs slightly from the current version in that it has two banks of 128 bits each (P and Q), rather than a single register file (M). This has the advantage that in a single PPEL instruction two bits from the register file can be accessed, but only if they have painstakingly been set up to use corresponding locations within the P and Q files. This idea was eventually dropped as not being general purpose enough, as well as rather cumbersome to use.

The original implementation included several implementation dependent I/O and debugging statements ('load' and 'save', 'display', 'print' and 'verbose', all described below). Some of these may reappear in future implementations of CPL-2, as and when appropriate.

The original user manual, giving details of the language and how to use the simulator, is appended below. Most of it does not make very exciting reading unfortunately, and it is mainly of historical interest. Tacked on the end of this appendix, however, are some timing results and sample output, which may be slightly more readily accessible.

The 'sim' interpreter

An emulator of the PPEL and ILP levels of the WPM architecture has been implemented, in the form of an interpreter for the Cluster Programming Language. A definition of the language, as implemented by *sim*, is given below.

This implementation of CPL is only suitable for SIMD or multi-SIMD processing over the whole image array, as it would need to be used in conjunction with some higher level language to provide a complete programming system for image understanding tasks. Its main use is for interactively developing and experimenting with low level algorithms.

Sim is written in C under Unix, and runs on both Vaxes and Suns. It currently implements 128x128 or 256x256 PPELs. It may be used interactively or from a script of CPL commands. Images to be processed are loaded and saved as Unix files, and may also be output on a graphic display. A basic timing metric is provided by displaying the execution time of a program in clock cycles.

CPL-1 language summary

A program consists of statements, one per line. Each line has the form

command arg1 arg2 ...

where these keyword and argument tokens are separated by white space characters or commas. If a token contains a space or comma it must be quoted (with a single quote). Blank lines and lines starting with a '#' are ignored as comments. The commands are summarised below, with more detailed notes on some of the commands and the form of parameters given in subsequent sections.

I/O and simulator control

load *filename n reg*	# load or save an image in a Unix file
save *filename n reg*	
display *n reg posn [label]*	# output PPEL array contents
verbose *[+−]mask*	# set simulator output level
print *arg, ...*	# display ILP variables or arguments
printf *str, arg, ...*	# formatted print
show *arg, ...*	# display internal simulator variables etc.
bitplane *name[[size]][=reg], ...*	# allocate a named array of PPEL memory
unsync	# force unsync'd operation
end	# terminate program

Basic PPEL operations

op i j d [addr] [?]	# single bit operation
repeat *n op i j d addr [?]*	# repeat *op i j d addr++* n times

Multi-bit PPEL operations

The following instructions may affect the X or F flag bits. The flag F is set when the result of a calculation is greater than 2^n-1 or less than 0. For non-arithmetic operations *n* defaults to 1.

clear *[n] dest [?]*	# dest ← 0
set *[n] dest [?]*	# dest ← all 1's
complement *[n] dest [?]*	# dest ← not dest
copy *[n] src dest [?]*	# dest ← src
copyc *[n] src dest [?]*	# dest ← not src
add *n src dest [?]*	# dest ← dest + src, F ← overflow
sub *n src dest [?]*	# dest ← dest − src, F ← overflow
subfrom *n src dest [?]*	# dest ← src − dest, F ← overflow
scale *n src dest const [?]*	# dest ← const × src, F ← overflow
accumulate *n src dest const [?]*	# dest ← dest + const × src, F ← overflow
compare *n reg1 reg2 [?]*	# test & set: X ← (reg1=reg2), F ← (reg1>reg2)

PPEL conditional operation

Any PPEL operation can be restricted to active PPELs (those with bit A set) if a '?' is appended to the instruction. Alternatively, one of the forms of '*where*' can be used to make all the PPEL statements in a block conditional on A. During a *where* block an individual operation can be made unconditional by appending a '!'.

'*Where*' statements may not be nested as such, but successive *where*'s can be used to progressively restrict the set of active PPELs, up to the occurrence of an 'endwhere'. (For the second and subsequent *where*'s 'A ← A & src', though '!' can still be used to force 'A ← src'. The old value of A is not saved.)

where *[!]src*	# A ← src, A ← not src
whereaddr *address*	# access a single PPEL by address 0..255
wheremask *hmask vmask*	# access PPELs using row/column masks
... stmts ...	
*[***elsewhere**	# A ← not A
... stmts ...]	
endwhere	

ILP operations

```
    if arg then                            # conditional (multi-)branching
      ... stmts ...
     [else
      ... stmts ...]
    endif

    for var = arg to arg [step arg]        # BASIC-style for..next loop
      ... stmts ...                        #  (unsynced if args aren't constants)
    next [var]                             #  var may be given for extra checking

    while arg do                           # standard while loop
      ...
    endwhile

    do                                     # ditto, but loop is carried out at least once
      ...
    while arg

    loop                                   # generalised loop
      ...
      exit [when arg]                      # can also break out of 'while' or 'for' loops
      ...
    endloop

    var := arg [binop arg]                 # assignment to ILP variable
    sync                                   # resynchronise the ILPs
```

Low level PPEL operations

The PPEL ALU performs any of the 2^{16} logic functions mapping three input bits (*i*, *j* and F) to two output bits (*d* and F), so the function opcode therefore has 16 bits (2 output bits for each of 8 different inputs).

For example, the code for the 'add with carry' function is 0001011001101011, which may be calculated by reading successive pairs out of the last two columns in the truth table below.

Input			i+j+F	
F	*j*	*i*	*F'*	*d*
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Table 5 Truth table for 'add with carry'

There are two ways of specifying this function opcode in CPL. It may be given explicitly, as a number consisting of eight base-4 digits preceded by an 'at' sign (eg. @01121223 for the example above). Alternatively, one of the predefined opcodes below may be used.

op	*d*	*f*	*op*	*d*	*f*
AND	i&j	f	NAND	~(i&j)	f
XOR	i^j	f	XNOR	~(i^j)	f
OR	i|j	f	NOR	~(i|j)	f
ADD	i^j	i&j	ADC	(i+j+f)%2	(i+j+f)≥2
SUB	(i-j)%2	j>i	SBC	(i-j-f)%2	(i-j-f)<0
SELI	i	f	SELJ	j	f
NOTI	~i	f	NOTJ	~j	f
SELIJ	i	j	CMP	~(i^j)	(i-j-f)<0
ORF	i|j|f	i|j|f	ANDF	i&j&f	i&j&f
SET	1	f	SETF	0	1
CLEAR	0	f	CLRF	0	0

i and *j* are PPEL operand addresses for the two source bits and *d* is the destination. They can be one of X, Y, Z, T, A, I, P, Q, N, NE, ... NW. The *addr* field specifies the address of the source or destination bit within the P or Q register file. Alternatively, extended PPEL addressing as below may be used (provided the operands remain consistent with the single address format of low level PPEL instructions), in which case the appropriate value for *addr* is filled in automatically. Some examples of valid PPEL instructions are:

```
AND X Y Z               # Z ← X&Y
SELI P P Y 3 ?          # if A then Y ← P3
ADC X Q17 P17           # P17 ← Q17+X+F, F ← carry
@02230223 Y Z X         # X ← Y&Z,  F ← Y|Z
```

Multi-bit PPEL operations

Arrays of PPEL registers are frequently used for multi-bit operations, eg. P10,11,12,13 may be regarded as an accumulator, containing one 4-bit number, to which the 4-bit value P14,15,16,17 may be added using the multi-bit macro instruction '*add 4 P14 P10*'.

Since it is inconvenient to keep track of these locations by explicit address, the language allows named bitplanes. The statement '*bitplane flag, image[8]*' would allocate a single bit register called *flag* and an 8-bit number register *image* in PPEL memory. The name *image* is used to refer to this 'variable' within multi-bit macro operations. Individual bits can be accessed using a subscript notation, eg. *image[7]* refers to the most significant bit (this works by adding an offset to the base address, so in fact *image* and *image[0]* are the same thing). The subscript may also come from an ILP variable, eg. *image[i]*.

Values can be copied from one of these variables to another, for example '*copy 8 image buffer*' copies *image[0]* to *buffer[0]*, *image[1]* to *buffer[1]*, and so on. '*scale 4 buffer Q10 7*' multiplies by a constant (Q10[0..3] ← 7 * buffer[0..3]), while '*add 8 1 image*' increments *image[0..7]* by one, and so on. Note that source and destination are always assumed to be of the same length, so if the destination needs to be longer to allow for the size of the result, the source has to be padded to the same length.

Operations between different variables of the same register file are implemented using an intermediate temporary store. If corresponding registers of the P and Q files are used, as in '*add 4 Q10 P10*', the operation can execute much faster. Placed bitplanes (eg. '*bitplane image[8]=P16*') may be used for this, but since this feature has not proven itself to be particularly useful it may disappear in the future (to be replaced with uniform addressing M0...M255).

Parameter summary

n is a bit count 1...32 (up to 8 bits for 'load' and 'save'); *reg* and *dest* are 'extended' PPEL register addresses or named bitplanes, eg. X, Y, F, NE, S, P16, Q0, Q127, image. The address may also be specified indirectly by the contents of an ILP variable, for example '%x'. *src* and *reg1,2* are similar but may additionally be prefixed by a direction as in N:P16 or SE:F, or they may be signed 16-bit constants (which may be the value of an ILP variable, eg. '$n').

Appropriate code for these different types of operand is generated by the compiler.

ILP operations

There are 26 ILP variables in this implementation. A *var* is one of the ILP variables **a** ... **z**. A *binop* is one of + – * / % & | ^ = != > < >= <= (all with the same meaning as in C).

arg is an ILP argument. It may be a *var*, a *const*, one of the special terms described below, or a simple expression containing these terms combined with *binop*'s, but note that expressions are evaluated strictly from left to right with no algebraic precedence of operators, nor are spaces allowed. For example, 'x+count/2' finds the average of *x* and *count*.

Special ILP terms include: **firstX** (address of first responding PPEL, coded as 16.x+y), **any** (Boolean with value 1 if any PPELs in the cluster are responding, ie. have their X bit set, or 0 otherwise), **count** (number of PPELs responding in this cluster). There are also several 'simulator constants': **myaddr** (the address of this ILP, from 0 to Isize×Isize – 1), **Isize** (number of ILPs across in the simulator), **Psize** (size of a cluster, PPELs across), and *&reg* (the address of a PPEL register); also 'time' (the current time in ticks for the ILP, but excluding time spent waiting for a sync) can be used.

read(*address,n,reg*) can be used to read data from a particular PPEL (in effect this works by using 'whereaddr' to select the PPEL, getting it to output its data onto the some/none bus one bit at a time, and accumulating the value in an ILP register).

Indexed PPEL addressing

Use of the PPEL operations has been described above with explicit arguments, but it is also possible to specify a variable or other parameter indirectly from the contents of an ILP variable.

Within an ILP expression the operator '&' can be applied to a PPEL operand (eg. X, N, SW, P0, image[7], N:Q16) to give the bit address used internally to refer to that PPEL register. Where one would use one of these registers in a PPEL instruction, the operator '%' can be used instead at that point to dereference an ILP variable which contains a PPEL address, eg. '*x := &P16; copy N:%x Y*' is equivalent to '*copy N:P16 Y*'. As a convenient abbreviation, one may omit the '&' when referring to directions, so '*for d = N to W step 2*' could be used to step through the four direct neighbours, for example. The loop variable can be used as a subscript directly (where N=0, NE=1, E=2 and so on) to accumulate a result over the neighbours.

One can specify numeric operands in PPEL operations (such as the bit count or a constant *src*) in a similar way, but using '$' (convert to numeric argument) rather than '%' (convert to address argument).

Saving images as Unix files

```
load filename n reg          # reg is a PPEL address
save filename n reg          # n is a bit count from 1 to 8
```

Images are stored as binary files, one byte per pixel, in rows from top to bottom (within each row the order is from left to right). We adopt the convention that the size of an image should be appended to the filename, so '*boat.128*' is an image at a resolution of 128×128, for example.

When loading a file the simulator first searches for a file with the name as given, then, if none is found, tries again with the current simulator resolution appended to the name. This allows one to write a resolution independent script: the appropriate image file is located automatically.

A filename may either contain an explicit path (including the use of '~/' to refer to the process's home directory), or be relative to the current directory. In the latter case the current directory is searched first, but if the image to be loaded is not found there then a standard images directory, with location given by the value of the environment variable 'IMAGES', is searched next.

The 'display' command

This outputs an image stored in the PPEL array. Unfortunately the arguments to 'display' are rather complicated, because there is a variety of ways of interpreting and displaying data.

> **display** *n reg posn [label]*

When the bit count *n* is from 1 to 8, that number of bits is read and displayed as a grey-scale (scaled to use the full range of available intensities regardless of the number of bits selected). If *n* is greater than 8, it is still interpreted as a bit count, with values up to 255 shown on a grey-scale and values over 256 mapped to one highlighted colour. If $n = -6$, six bits are used to display a colour image (two bits each of RGB). When $n = -9$ or -10, the bottom 8 bits are mapped to a grey scale and the upper 1 or 2 bits are used as a mask plane for highlighting ($0 \rightarrow$ grey scale, $1 \rightarrow$ red, $2 \rightarrow$ yellow, $3 \rightarrow$ purple). With $n = -8$ the raw colour map is used (beware!)

If 256 is added to *n*, or if '*display i ...*' is used, the intensity display is inverted when hardcopy output is being generated (this should be used when displaying edge intensities for example, because in this case white on the screen corresponds better to black on paper).

The *posn* argument normally lies between 0 and however many images fit on the screen (less one). This number depends on the output device: for the Sun it is currently 12. If *posn* = 100 no output is generated, and if *posn* is 101 or greater (*posn* – 100) lines of textual numeric output are produced on the terminal (useful for looking at exact values for debugging). The *label* is a string of up to 20 characters (or possibly more, depending on output device).

Two forms of graphical output are currently supported: display on a colour Sun, or PostScript output for producing hardcopy on a laser printer.

On the Sun the Sunview graphics system is used, creating a window/frame for the output, which must be closed manually by selecting the menu option 'Quit' with the mouse before the simulator can terminate.

The PostScript version sends its output to *stdout*, and so is unsuitable for interactive use (its output should be piped through '*lpr -Ppsc*'). It sets '*verbose = 0*', which means that all other output is suppressed, other than error messages, which are sent to *stderr*.

It is also possible to select textual output, displaying the output of three rows of the first ILP numerically with the command line option '-t'.

The 'print' and 'show' commands

The 'print' command displays the value of ILP variables and expressions (including parameters such as **count** or **firstX**). The command takes a variable number of arguments, and the output may be annotated if a string in double quotes is given (eg. '*print "count=", count*'). The expressions are printed out using a separate line for each ILP, so it may be helpful to print **myaddr** to identify which ILP each line of output refers to.

'Printf' is a formatted print, taking a format string in the manner of *printf(3)* in the Unix standard library.

'Show' is used to display internal simulator variables: **active**, a mask showing which ILPs are enabled and what the current ILP is when running out of sync; **time**, the simulation time in ticks for the ILPs, where the subtotal for the current simulation slice is shown in parentheses; **symbols**, the bitplane symbol table; or the current **verbose** setting. The 'show' command can also be used on the parameters accepted by 'print' to print the values for all the ILPs on one line (when they are running in sync); or to display the value used internally to refer to PPEL addresses etc.

The 'verbose' command

This selects the amount of simulator output. Its argument is a bit field, where the different bits (hex values shown) have the following meaning (the name used internally is also shown):

1	DPRTEN	'Print' command enabled (†)
2	DTOTAL	Display total number of ticks at end of program (†)
4	DWARN	Display warning messages (eg. divide by 0) (†)
8	DINFO	Display normal informational messages (†)
10	DVERB	More verbose (extra info and status messages) (*)
20	DTRACE	Trace internal simulator calls (*)
40	DILP	Trace low-level calls to '*ilp*'
80	DDEBUG	Simulator debugging
100	DTEACH	Display time taken by each instruction in ticks (‡)
200	DECHO	Echo each instruction (‡)
400	DTEXT	Text output only (graphics disabled)

The default outputs are marked (†). This default may be changed by giving a command line option when the simulator is invoked. To echo statements as they occur when an interpreter script is running the option -e should be selected, this adds the (‡) options. The command line option -v adds the (*) outputs, -t selects text output (and only one ILP is simulated so the simulator runs faster with this option), -i adds 'ilp' tracing, and -s disables all output other than fatal error messages.

The simulator routine *ilp( )* performs a single ILP or PPEL operation on all the currently active ILPs/clusters, so tracing calls to this routine with -i would show all the single bit operations carried out during a multi-bit macro, for example.

'Verbose' can either take an absolute parameter in hex, or a change from the current value if prefixed with '+' or '–' to set or reset particular bits, eg. '*verbose +80*' adds simulator debugging output.

Use of the simulator

The interpreter may be run interactively or from a command script, depending on whether a *cmdfile* is specified:

sim [-eistTv?] *[cmdfile]*

The command line options select the amount of diagnostic information that gets output by the simulator (see description of the 'verbose' command, or type 'sim -?' for a brief description of the options).

Unix lets you make a script directly executable if you have '#!sim' as the first line of the file and make it executable (with 'chmod +x').

The interpreter terminates when it gets to the end of file, or upon reading an '*end*' instruction, and then prints out the total execution time of the program in *clock ticks*, where a tick is the time taken to do one PPEL instruction (typically 100 ns).

The keyboard interrupt character (usually ^C) can also be used to terminate a simulator run. When the interpreter is being used interactively ^C should be pressed once to interrupt an instruction and return to the input prompt '>', or twice in succession to quit.

Annotated listings

The '-T' option can be used to generate additional timing output. This is merged together with the program source file using the awk script 'annotate' to produce an annotated listing showing the average time spent executing each instruction. Usage: 'sim -T script >output' followed by 'annotate script output'.

Unsynced ILP operation

Normally all the ILPs run in lock-step synchronisation, but after a conditional statement such as '**if**' is encountered, multiple independent execution branches are followed until all the ILPs **sync** themselves again. The simulator implements this by doing multiple passes through the program, simulating one ILP to completion (or to the next **sync** instruction) and then the next and so on, keeping track of the time and current 'program counter' for each ILP. (This only works from simulator scripts, as opposed to interactive use, as it is not possible to 'rewind' standard input.)

The **unsync** command forces this mode of operation and is occasionally useful to overcome limitations in the current simulator implementation (for example, if you supply a PPEL parameter from an ILP variable and the simulator is running in sync at that point the value is taken from ILP 0, so if the ILPs have different values you will get incorrect results).

The inputs from neighbouring PPELs along the edge of a cluster are undefined when the simulator is running in unsynced mode. This is probably a sort of bug, but currently very difficult to 'fix'. A neat solution might be to always feed back a PPELs own output along the edge.

Some timing results

A number of common low level image processing algorithms that might be performed at the PPEL and ILP levels have been coded up in CPL and run on the simulator. The timings given below assume one pixel per PPEL and a 100ns ILP/PPEL cycle time.

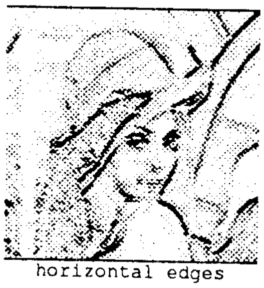
Local maximum	8 μs
Local mean	6 μs
Laplacian operator	8 μs
Sobel edge detection	24 μs
Guarded thinning	63 μs
Chain coding and output	395 μs
Arbitrary 5×5 convolution	180 μs
64 bin histogram	65 μs

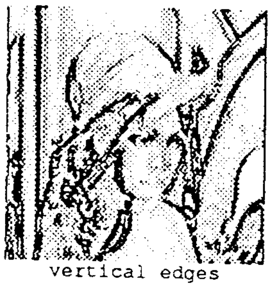
Table 6 Simulator timings

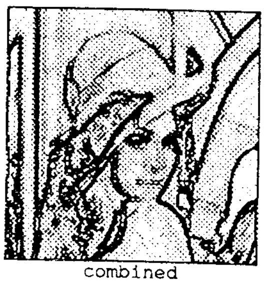
Example of old-style CPL code, together with simulator output

```
#!sim
# Sobel edge detection convolution

# -1 -2 -1     -1  0  1
#  0  0  0     -2  0  2
#  1  2  1     -1  0  1

bitplane image[11]=P16, partial[11]=Q16
bitplane sobel[11], result[11]

load 'girl' 8 image
display 8 image 0 'original image'

# partial(10) := image*2+E+W
copy 8 image partial[1]
add 10 E:image partial
add 10 W:image partial

# result(11) := S:partial-N:partial
copy 10 S:partial result
sub 11 N:partial result

# negate if negative to get absolute value
where F
   subfrom 11 0 result
endwhere
# sobel used as main accumulator for |Gx| + |Gy|
copy 10 result sobel

# Display result
display i 10 result 1 'horizontal edges'

# Same thing vertically
clear partial[0]
copy 8 image partial[1]
clear 2 image[8]
add 10 N:image partial
add 10 S:image partial

clear result[10]
copy 10 E:partial result
sub 11 W:partial result
where F
   subfrom 11 0 result
endwhere
display i 10 result 2 'vertical edges'
add 11 result sobel

display i 11 sobel 3 'combined'
```

original image


horizontal edges


vertical edges


combined

```
Total simulator run time: 235 ticks
```